Rust Book 10.2: Traits - Defining Shared Behavior

#rust Traits: Defining Shared Behavior

A **trait** tells the Rust compiler about functionality a particular type has and can share with other types.

We can use traits to define shared behavior in an abstract way. We can use trait bounds to specify that a generic can be any type that has certain behavior.

Note: Traits are similar to a feature often called **interfaces** in other languages, although with some differences.

Defining a Trait

Defining a Trait

A type's behavior consists of the methods we can call on that type. Different types share the same behavior if we can call the same methods on all of those types. Trait definitions are a way to group method signatures together to define a set of behaviors necessary to accomplish some purpose.

Filename: src/lib.rs

```
pub trait Summary {
    fn summarize(&self) -> String;
    // NO METHOD IMPLEMENTATION
    // just the signature and a ;
}

pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}, by {} ({})", self.headline, self.author, self.location)
```

```
    }
}

pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

TheSummarytrait would also need to be a public trait for another crate to implement it, which it is because we put thepubkeyword beforetraiti

One restriction to note with trait implementations is that we can implement a trait on a type only if either the trait or the type is local to our crate.

We can't implement external traits on external types.

This restriction is part of a property of programs called **coherence**, and more specifically the **orphan rule**, so named because the parent type is not present. This rule ensures that other people's code can't break your code and vice versa. Without the rule, two crates could implement the same trait for the same type, and Rust wouldn't know which implementation to use.

Default Implementations

```
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more…)")
    }
```

```
}
```

To use a default implementation to summarize instances ofNewsArticleinstead of defining a custom implementation, we specify an emptyimplblock withimpl Summary for NewsArticle {}.

Creating a default implementation for `summarize` doesn't require us to change anything about the implementation of `Summary` on `Tweet`. The reason is that the syntax for overriding a default implementation is the same as the syntax for implementing a trait method that doesn't have a default implementation.

Default implementations can call other methods in the same trait, even if those other methods don't have a default implementation.

In this way, a trait can provide a lot of useful functionality and only require implementors to specify a small part of it.

```
pub trait Summary {
    fn summarize_author(&self) -> String;

    fn summarize(&self) -> String {
        format!("(Read more from {}...)", self.summarize_author())
    }
}
```

Traits as arguments

Traits as arguments

Now that you know how to define traits and implement those traits on types, we can explore how to use traits to accept arguments of many different types.

```
pub fn notify(item: impl Summary) {
    println!(“Breaking news! {}”, item.summarize());
}
```

Trait Bounds

The `impl Trait` syntax works for short examples, but is syntax sugar for a longer form. This is called a **trait bound**, and it looks like this:

```
pub fn notify<T: Summary>(item: T) {
    println!("Breaking news! {}", item.summarize());
}
```

This is equivalent to the example above, but is a bit more verbose. We place trait bounds with the declaration of the generic type parameter, after a colon and inside angle brackets. Because of the trait bound onT, we can callnotifyand pass in any instance ofNewsArticleorTweet. Code that calls the function with any other type, like aStringor ani32, won't compile, because those types don't implementSummary.

Specify multiple traits with+

```
pub fn notify(item: impl Summary + Display) {
```

This syntax is also valid with trait bounds on generic types:

```
pub fn notify<T: Summary + Display>(item: T) {
```

`where` clauses for clearer code

Rust has alternate syntax for specifying trait bounds inside a `where` clause after the function signature. So instead of writing this:

```
fn some_function<T: Display + Clone, U: Clone + Debug>(t: T, u: U) -> i32 {
```

we can use a `where` clause, like this:

```
fn some_function<T, U>(t: T, u: U) -> i32
    where T: Display + Clone,
          U: Clone + Debug
```

```
{
```

Returning Traits

Returning Traits

We can use the `impl Trait` syntax in return position as well, to return something that implements a trait:

```
fn returns_summarizable() -> impl Summary {
    Tweet {
        username: String::from(“horse_ebooks”),
        content: String::from(“of course, as you probably already know,
people”),
        reply: false,
        retweet: false,
    }
}
```

This signature says, “I’m going to return something that implements the `Summary` trait, but I’m not going to tell you the exact type.” In our case, we’re returning aTweet, but the caller doesn’t know that.

…two features that rely heavily on traits: **closures**, and **iterators**. These features create types that only the compiler knows, or types that are very, very long. `impl Trait` lets you simply say “this returns an `Iterator`” without needing to write out a really long type.

This only works if you have a single type that you’re returning

Fixing thelargestFunction with Trait Bounds

```
error[E0369]: binary operation `>` cannot be applied to type `T`
 --> src/main.rs:5:12
  |
5 |         if item > largest {
  |            ^^^^^^^^^^^^^^
  |
```

```
  = note: an implementation of `std::cmp::PartialOrd` might be missing for `T`
```

```
// fn largest<T>(list: &[T]) -> T {
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }
    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest(&char_list);
    println!("The largest char is {}", result);
}
```

Compiler message: "note: move occurs because `item` has type `T`, which does not implement the `Copy` trait"

```
error[E0508]: cannot move out of type `[T]`, a non-copy slice
 --> src/main.rs:7:23
  |
7 |     let mut largest = list[0];
  |                       ^^^^^^^
  |                       |
```

```
  |                       cannot move out of here
  |                       help: consider borrowing here: `&list[0]`

error[E0507]: cannot move out of borrowed content
 --> src/main.rs:9:18
  |
9 |     for &item in list.iter() {
  |         -----    ^^^^^^^^^^^ cannot move out of borrowed content
  |         ||
  |         |data moved here
  |         help: consider removing the `&`: `item`
  |
note: move occurs because `item` has type `T`, which does not implement the
`Copy` trait
 --> src/main.rs:9:10
  |
9 |     for &item in list.iter() {
  |          ^^^^
```

The key line in this error is `cannot move out of type [T], a non-copy slice`. With our non-generic versions of the `largest` function, we were only trying to find the largest `i32` or `char`. Types like `i32` and `char` that have a known size can be stored on the stack, so they implement the `Copy` trait. But when we made the `largest` function generic, it became possible for the `list` parameter to have types in it that don't implement the `Copy` trait. Consequently, we wouldn't be able to move the value out of `list[0]` and into the `largest` variable, resulting in this error.

If we don't want to restrict the `largest` function to the types that implement the `Copy` trait, we could specify that `T` has the trait bound `Clone` instead of `Copy`. Then we could clone each value in the slice when we want the `largest` function to have ownership. Using the `clone` function means we're potentially making more heap allocations in the case of types that own heap data like `String`, and heap allocations can be slow if we're working with large amounts of data.

Another way we could implement `largest` is for the function to return a reference to a `T` value in the slice. If we change the return type to `&T` instead of `T`, thereby changing the body of the function to return a reference, we wouldn't need the `Clone` or `Copy` trait

bounds and we could avoid heap allocations.

```
fn largest<T>(list: &[T]) -> &T
    where T: PartialOrd
{
    let mut largest = &list[0];
    for item in list.iter() {
        if item > largest {
            largest = &item;
        }
    }
    &largest
}
```